Exercises for

Database Implementation

Elite Graduate Program Software Engineering

Andreas Kipf (kipf@in.tum.de)

Assignment 2

Info

- Please submit by 5 May 2015, 09:00am.
- Your submission should include your implementation and testing code as well as information on how to build your project, e.g. a `Makefile`. It should also contain a `README` file explaining how to run the program and how you tested it. Example directory structure:

```
BufferManager.zip
 |-yourFancyDB
    |-Makefile
    |-README
    |-buffer
    |  |-BufferManager.hpp
    |  |-BufferManager.cpp
    |  |-BufferFrame.hpp
    |  |-BufferFrame.cpp
    |-testing
       |-BufferManagerTest.cpp
```

  Please do not include binary files in your submission and ensure that your program can be built and run on Linux.

- Please include your full name and the full name of your teammate(s) in your email an put them on CC.
- You are encouraged to use C++ for the exercises.

Excercise 1

Write a basic buffer manager that manages buffer frames and controls concurrent access to these frames. The buffer manager should offer the following functionality:

`BufferManager::BufferManager(uint64_t size)` Create a new instance that keeps up to `size` frames in main memory.

`BufferFrame& BufferManager::fixPage(uint64_t pageId, bool exclusive)` A method to retrieve frames given a page ID and indicating whether the page will be held exclusively by this thread or not. The method can fail (by throwing an exception) if no free frame is available and no used frame can be freed. The page ID is split into a segmentID and the actual page ID. Pages are stored on disk in files with the same name as the segment ID (e.g. `1`).

`void BufferManager::unfixPage(BufferFrame& frame, bool isDirty)` Return a frame to the buffer manager indicating whether it is dirty or not. If dirty, the page manager must write it back to disk. It does not have to write it back immediately, but must not write it back before `unfixPage` is called.

`void* BufferFrame::getData()` A buffer frame should offer a method giving access to the buffered page.[1] Except for the buffered page, `BufferFrame` objects can also store control information (page ID, dirtyness, ...).

`BufferManager::~BufferManager()` Destructor. Write all dirty frames to disk and free all resources.

Your buffer manager should have the following features:

- High performance. Release locks as early as possible.
- Concurrency: It should be able to handle concurrent method invocations efficiently (e.g. using latches[2]). Requests to `fixPage` should block until the requested access (exclusive or shared) can be fulfilled.
- Buffering: It should use a buffer of `size` frames to keep pages in memory as long as possible. If no free frames are available, old frames should be reclaimed using the 2Q strategy. Describe your 2Q implemantation in the `README` or in comments.
- The page size should be a (constant) multiple of the size of a virtual memory page (4096 bytes on most systems).

Your buffer manager does not need to have the following advanced features[3]:

- Asynchronous flushing of pages to disk.

[1] Note that the pointer loses validity if `unfixPage` is called on the frame. A nicer, more robust way to handle this might be to employ move semantics (see `std::unique_ptr`).
[2] e.g. `pthread_rwlock_t`
[3] But you may implement them, of course!

- Prefetching of pages that are likely to be accessed in the near future.

Excercise 2
Use the test program from the website to validate your implementation.